Parallel Programming Archetypes in Combinatorics and Optimization

Thesis by
Svetlana A. Kryukova
Advisor: K. Mani Chandy

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

California Institute of Technology
Computer Science Department
Pasadena, California 91125

June 12, 1995

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
12 JUN 1995		**12-06-1995 to 12-06-1995**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Parallel Programming Archetypes in Combinatorics and Optimization	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Air Force Office of Scientific Research,875 North Randolph Street Suite 325,Arlington,VA,22203-1768**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**56**	

Abstract

A Parallel Programming *Archetype* is a language-independent program design strategy. We describe two archetypes in combinatorics and optimization, their components, implementations, and example applications developed using an archetype.

Acknowledgements

Many thanks to Mani Chandy whose guidance and encouragement as my research advisor was very much appreciated.

Also I would like to thank the other members of Compositional Systems Research Group — Berna Massingill, Adam Rifkin, Eve Schooler, Paul Sivilotti, John Thornley — for valuable discussions, comments and suggestions.

I wish to thank Caltech and Green Hills for supporting my education.

Last, but not least, I would like to thank my husband, Anatoly, for his loving support.

This research was supported in part by AFOSR grant F49620-94-1-0244 and by the NSF under Cooperative Agreement No. CCR-9120008.

Contents

1 Introduction **1**
- 1.1 Goal . . . 1
- 1.2 Motivation . . . 1
- 1.3 What Are Archetypes? . . . 2
- 1.4 Architectures, Languages and Libraries . . . 2
- 1.5 Overview . . . 3

2 The Divide-and-Conquer Archetype **5**
- 2.1 Introduction . . . 5
- 2.2 Archetype Skeleton . . . 6
- 2.3 Archetype Components . . . 6
  - 2.3.1 Algorithmic Parameters . . . 8
- 2.4 Approaches to Parallel Implementation . . . 9
- 2.5 Data Flow Approach and Performance . . . 12
  - 2.5.1 Mapping . . . 13
  - 2.5.2 Granularity . . . 14
  - 2.5.3 Skeleton . . . 17
- 2.6 Implementations . . . 19
- 2.7 Applications . . . 23
  - 2.7.1 Mergesort . . . 23

3 The Branch and Bound Archetype **28**
- 3.1 Introduction . . . 28
  - 3.1.1 Assumptions . . . 29
- 3.2 Archetype Skeleton . . . 29
- 3.3 Archetype Components . . . 30
- 3.4 Approaches to Parallel Implementation . . . 32
- 3.5 Implementations . . . 33

3.6 Applications 38
3.6.1 Zero-One Knapsack 38

4 Conclusion **43**

A Electronic Textbook **45**

Bibliography **46**

List of Figures

2.1 Data flow graph of a Divide-and-Conquer algorithm. 9
2.2 A mapping of a Divide-and-Conquer data flow graph. 13
2.3 Group processes. 17
2.4 Execution time of finding the minimum element. 18
2.5 Speedup of parallel mergesort on Touchstone Delta. 27

3.1 Execution time of zero-one knapsack program on Touchstone Delta. 41

Chapter 1

Introduction

1.1 Goal

The goal of this thesis is to study two *archetypes* in combinatorics and optimization, the Divide-and-Conquer Archetype and the Branch and Bound Archetype, and to demonstrate how these archetypes can be used for the systematic design of efficient sequential and parallel programs. The research whose results are presented in this document is part of the ongoing project on Parallel Programming Archetypes.

1.2 Motivation

As networks of workstations and distributed systems that can exploit parallel computing become more widespread, the need for tools to aid the development of parallel programs grows as well. Many scientists do not take advantage of the available hardware because often the effort required for developing a new parallel program from scratch or parallelizing existing code is not justified by the potential speedup. We think that it is possible to reduce this effort by using parallel programming *archetypes*.

Many parallel applications share common features in design, program structure, communication pattern, reasoning, debugging, testing and performance tuning. This allows for development of an *archetype* as an abstraction that embodies these common characteristics. Understanding of an archetype helps a programmer to understand all algorithms contained within the archetype's application domain. Moreover, knowledge of several archetypes forms a basis for a systematic approach to program design for a

new problem.

Archetypes are being developed as language-independent design methods, allowing users to choose programming languages and communication libraries, as well as familiar programming environments and debugging tools. For many archetypes most of the debugging and testing can be done on a sequential program, from which a correct and efficient parallel program can be easily derived.

1.3 What Are Archetypes?

An *archetype* is a method of problem solving characterized by a design strategy. It consists of several parts including:

1. the structure of a class of programs,
2. methods of developing parallel and sequential applications,
3. frameworks for reasoning about correctness,
4. suggestions for test suites and debugging, and tips based on the experience of others,
5. suggestions for performance tuning and performance models for different architectures, and
6. a collection of applications developed using an archetype, each with a collection of programs in several programming languages.

Currently archetypes are being developed in two areas: archetypes for scientific applications [5, 10] such as the Mesh Archetype and the Spectral Methods Archetypes, and archetypes for combinatorics and optimization, such as the Branch and Bound Archetype and the Dynamic Programming Archetype.

1.4 Architectures, Languages and Libraries

Today many different architectures are available to the users for development of concurrent programs: heterogeneous or homogeneous networks of workstations or PCs, massively parallel supercomputers and multiprocessor workstations. These architectures have different characteristics and may require different programming approaches in order to achieve good performance. In

order to show how the archetypal approach can be used for different architectures, we will consider two major architectures: networks of single-processor computers, such as workstations or PCs, and a supercomputer. The former can be characterized by low communication to computational speed ratio, whereas the latter can be characterized by high communication speed. Most programs in this report were written for a network of Sun SPARC stations and for the Intel Touchstone Delta.

Archetypes are being developed as a language-independent approach to sequential and parallel program design, allowing programmers to take advantage of the special features of parallel languages and communication libraries. In this report we used C programming language together with two of communication libraries and a task-parallel language:

PVM: Programs developed for networks of workstations were written using the Parallel Virtual Machine (PVM) system developed at Oak Ridge National Laboratory and the University of Tennessee [7]. The system uses message passing to exploit parallel computing across a wide variety of distributed systems, including networks of workstations and massively parallel computers.

NX: Programs developed for the Intel Touchstone Delta were written in the C programming language using the NX communication library for message-passing between processes [8].

CC++: Several programs were written in Compositional C++ (CC++), a parallel programming language based on the C++ programming language and developed at Caltech [3, 12]. CC++ has a few simple extensions to allow construction of parallel libraries on a variety of architectures.

1.5 Overview

This report is divided into 2 chapters, each of which describes a different programming archetype. Each chapter contains the following sections:

Introduction: an intuitive description of an archetype's approach and programs in the archetype's application domain.

Archetype Skeleton: the archetype's strategy, presented in a more formal way, with the skeleton of the main procedure.

Archetype Components: the specification of user-defined functions, procedures and data types.

Approaches to Parallel Implementation: one or more approaches to parallel implementation of an archetype.

Implementations: description of several implementations of the archetype's programming template.

Applications: description of one or more applications from an archetype's domain.

The following points concerning assertional and coding notation should be noted:

- All data type names used throughout the report end with "`_t`".
- In some cases, the names of the procedures are also used as predicates in the assertions. Such a predicate holds if and only if the execution of the procedure with the given input parameters produces the given output parameters. For example, if procedure `proc(x, y)` has input parameter `x` and output parameter `y`, then predicate $proc(a, b)$ holds if and only if, after execution of `proc(`a`,y)`, `y =` b.

Chapter 2

The Divide-and-Conquer Archetype

2.1 Introduction

The Divide-and-Conquer Archetype is based on a well-known strategy [2, 11] for solving large problems, if there exists an algorithm for solving smaller problems of the same type. Although both recursive and non-recursive implementations of this approach exist, it is usually described as the following recursive algorithm: when given a large problem,

1. take the problem and divide it into *strictly* smaller problems of the same type; continue dividing until a problem is reached that is small enough to be solved directly. A problem of such size is usually called a *base-case* problem, or simply a *base-case*;

2. upon reaching a base-case problem, solve it using some known algorithm;

3. then take the solutions to the smaller problems and merge them into the solutions to larger problems, until a solution to the original problem is obtained.

Whether this approach will produce an efficient sequential algorithm depends on the ability to split a problem and/or recombine subsolutions in efficient manner. However, even for problems that do not have efficient sequential solutions, using the Divide-and-Conquer strategy, the archetype might provide an efficient parallel solution.

2.2 Archetype Skeleton

The outline of the sequential Divide-and-Conquer Archetype is as follows:

```
Solution_t DnC(Problem_t aProblem)
{
  Solution_t aSolution;

  if (Size(aProblem) ≤ BaseCaseSize)
    aSolution = BaseCaseSolution(aProblem);
  else {
    Problem_t subProblems[K];
    OtherInfo_t Other;
    Solution_t subSolutions[K];
    int i;

    Split(aProblem, subProblems, Other);
    for (i = 0; i < K; i++)
      subSolutions[i] = DnC(subProblems[i]);
    aSolution = Merge(subSolutions, Other);
  }
  return aSolution;
}
```

Constants `K` and `BaseCaseSize`, and functions `Size()`, `BaseCaseSolution()`, `Split()` and `Merge()` are described in the following sections.

2.3 Archetype Components

In general, in order to develop a sequential Divide-and-Conquer algorithm, the user has to define 3 data types and several functions and procedures on these data types. Additionally, for specification and reasoning purpose, several predicates should be defined.

Data types:

`Problem_t` is a user-defined data type representing a problem being solved.

`Solution_t` is a user-defined data type representing a solution to some problem of type `Problem_t`.

`OtherInfo_t` is an optional user-defined data type representing some information which is produced by the `Split()` procedure in addition to the subproblems, and then used by the `Merge()` procedure. Many Divide-and-Conquer algorithms do not use this data type.

Predicates: For specification and reasoning purposes only, the user should define two predicates on the variables of the user-defined data types:

$isWellFormed(P)$ holds if and only if problem P is a well formed problem. The reason for having this predicate is that programming languages allow variables of certain types to have many values, some of which are not well formed in the context of the problem being solved.

$isSolution(P, S)$ holds if and only if solution S is a correct solution to problem P.

Constants, Functions and Procedures: The following procedures and functions must be defined for a Divide-and-Conquer algorithm according to the specifications given:

`Size()` is a function that returns some metric value on the size of a problem. Together with the constant `BaseCaseSize` this function is used to determine whether a given problem is small enough to be solved directly.

`BaseCaseSize` is a constant that defines the largest problem that can be solved directly (using some other algorithm). Any problems that have size smaller than `BaseCaseSize` will be solved using some other algorithm.

`BaseCaseSolution()` is a function that uses some algorithm to solve directly the problems of the size no larger than `BaseCaseSize`. The formal specification of this function is as follows:

```
Solution_t BaseCaseSolution(Problem_t aProblem)
/* Precondition:  isWellFormed(aProblem) and
 *    Size(aProblem) ≤ BaseCaseSize
 * Postcondition: (Return value = aSolution) and
 *    isSolution(aProblem, aSolution)
 */
```

`Split()` is a function that divides a given (large) problem into K subproblems of *strictly* smaller size, whose solutions when merged produce the solution to the given problem. Formally,

```
void Split(Problem_t aProblem, Problem_t subProblems[],
           OtherInfo_t Other)
/* Precondition:  isWellFormed(aProblem) and
 *    Size(aProblem)  >  BaseCaseSize
 * Postcondition:
 *    (∀k : 0 ≤ k < K :  Size(subProblems[i]) < Size(aProblem))
 *    and
 *    (∃Sols[] :
 *       (∀k : 0 ≤ k < K : isSolution(subProblems[k], Sols[k])) :
 *       (∀S : isSolution(aProblem, S) : S = Merge(Sols, Other)))
 */
```

`Merge()` is a function that merges a given set of subsolutions into the solution to the bigger problem, such that the subproblems, to which the given subsolutions are solutions, were produced by the split of that problem. Formally:

```
Solution_t Merge(Solution_t subSolutions[],
                 OtherInfo_t Other)
/* Precondition:  true
 * Postcondition: (Return value = aSolution) and
 *    (∃ Probs[] :
 *       (∀k : 0 ≤ k < K :
 *          isSolution(Probs[k], subSolutions[k])) :
 *       (∀P : Split(P, Probs[], Other) :
 *          isSolution(P, aSolution))
 */
```

2.3.1 Algorithmic Parameters

In the performance analysis we will use the following parameters of a Divide-and-Conquer algorithm:

K is the maximum number of the subproblems returned by the `Split()` procedure, and

S/b is the maximum size of the subproblems returned by the `Split()`, where S is the size of the original problem.

For example, for the Mergesort algorithm these parameters are $K = 2$ and $b = 2$, for Binary search $K = 1$ and $b = 2$.

2.4 Approaches to Parallel Implementation

The data flow structure of a Divide-and-Conquer algorithm is shown in figure 2.1. It consists of a growing tree of `Split()` processes concatenated with a shrinking tree of `Merge()` processes. Two main approaches to parallel implementation of a Divide-and-Conquer algorithms map this graph differently in time and space.

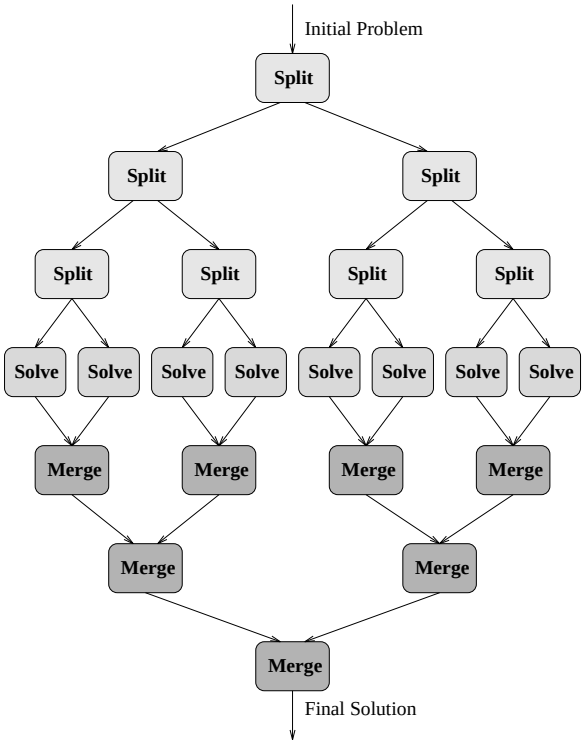

Figure 2.1: Data flow graph of a Divide-and-Conquer algorithm.

Data Flow Approach. If the subproblems produced by the `Split()` function are independent and can be solved separately, then one can parallelize the solving of subproblems in the sequential algorithm (making a `parfor` loop out of the `for` loop in the `DnC()` procedure in section 2.2). Thus, `Split()`, `Merge()` and the solving of the subproblems will be executed sequentially, with the subproblems solved in parallel. The general program structure of this approach is given below:

```
Solution_t Parallel_DnC1(Problem_t aProblem)
{
  Solution_t aSolution;
  if (Size(aProblem) ≤ BaseCaseSize)
    aSolsution = BaseCaseSolution(aProblem);
  else {
    Problem_t subProblems[K];
    Solution_t subSolutions[K];
    OtherInfo_t Other;
    int i;

    Split(Problem, subProblems, Other);
    parfor(i = 0; i < K; i++)
      subSolutions[i] = Parallel_DnC1(subProblems[i]);
    aSolution = Merge(subSolutions, Other);
  }
}
```

Note that the structure of this approach does not require any changes to the user-defined sequential functions `Size()`, `BaseCaseSolution()`, `Split()` and `Merge()`.

Control Flow Approach. This approach is applicable to Divide-and-Conquer algorithms for which a problem and/or a solution can be represented as a collection of independent parts of the same type. All component functions of the archetype then can be rewritten as separate processes which use messages to communicate parts of subproblems/subsolutions to each other, and which produce parts of subproblems/subsolutions based only on partial information (several parts of a subproblem/subsolution) received from the other processes. Then, after the decision is made whether a problem is a base-case problem or not, function `Split()`,

function Merge() and the solving of the subproblems can be executed concurrently. The skeleton for this implementation is as follows:

```
Solution_t Parallel_DnC2(Problem_t aProblem)
{
  Solution_t aSolution;
  if (Size(aProblem) ≤ BaseCaseSize)
    aSolution = BaseCaseSolution2(Problem);
  else {
    Problem_t subProblems[K];
    Solution_t subSolutions[K];
    OtherInfo_t Other;

    par{
        Split2(Problem, subProblems, Other);
        parfor(int i = 0; i < K; i++)
          subSolutions[i] = Parallel_DnC2(subProblems[i]);

        aSolution = Merge2(SubSolutions, Other);
      }
  }
}
```

Note that the declaration of subProblems and subSolutions as well as the parameters to the functions corresponds more to the declarations of the communication channels between the processes rather than to the variable declarations. Also note that we use function names of the form FunctionName2() instead of FunctionName() to indicate that these functions differ from those used in the sequential algorithm.

Even though the Control Flow approach allows for more concurrency, the Data Flow approach is very easy to use to parallelize already existing sequential code. Even development of a parallel algorithm from scratch is easier with this approach, since the user can develop and debug the sequential program first. The rest of this chapter focuses on the implementation and performance analysis of the Data Flow approach.

2.5 Data Flow Approach and Performance

Once the user has developed a sequential Divide-and-Conquer algorithm, the Data Flow approach can easily be used to obtain a parallel algorithm from the sequential code. A programming template can be provided to the user to obtain a parallel algorithm by simple instantiation of the component sequential functions. However, in order for the parallel implementation to be efficient several issues have to be taken into consideration, such as granularity and mapping of the processes.

To analyze and predict the performance of a parallel Divide-and-Conquer algorithm, we need to know some performance characteristics of the target architecture:

- N_p: the number of the processors in the system.
- T_{com}: average time for communication between the processors or machines in the system. This time can be given as a function of the size of the message or the size of a problem.
- T_{split}: execution time of the sequential `Split()` procedure as a function of the problem size.
- T_{merge}: execution time of the `Merge()` procedure as a function of the problem size.
- $\mathrm{T}_{base-case}$: execution time of the `BaseCaseSolution()` as a function of the problem size.

We also make the following assumptions about the target architecture:

- available processors are identical;
- only asynchronous communication actions are used; in particular, only non-blocking sends are used;
- the time that the sender of a message spends on communication actions is negligibly small.

Assuming that the size of the original problem is $S = b^q$ and the size of a base-case problem is `BaseCaseSize` $= 1$, and using the above functions, we can express the execution time of the sequential algorithm as follows:

$$\mathrm{T}^{seq}(b^q) = \sum_{i=0}^{q-1} K^i \left(\mathrm{T}_{split}(b^{q-i}) + \mathrm{T}_{merge}(b^{q-i}) \right) + K^q \mathrm{T}_{base-case} \qquad (2.1)$$

2.5.1 Mapping

Let us consider a mapping of the data-flow graph of a Divide-and-Conquer algorithm (e.g., see figure 2.1) onto the processors of the system.

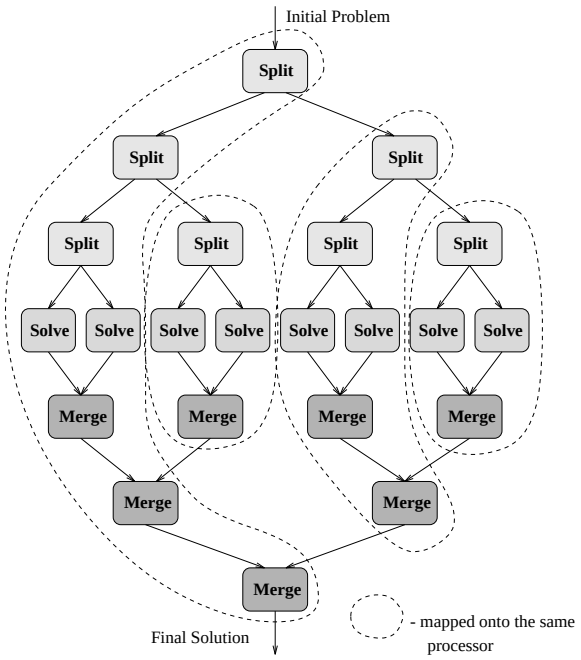

Figure 2.2: A mapping of a Divide-and-Conquer data flow graph.

Note that the execution of a process at any level of the data-flow graph does not start until all preceding processes (predecessors in the graph) have terminated. If the sizes of the subproblems returned by the `Split()` functions are approximately equal, then the execution times of all processes on one level of the graph are equal. Then, suppose that we map the processes onto the system as shown in figure 2.2. If the communication overhead is relatively small, such a mapping will produce an efficient implementation.

From the execution time functions for the `BaseCaseSolution()`, `Split()` and `Merge()` procedures, we can derive a recursive expression for the execution time of the parallel algorithm with this method:

$$\mathrm{T}^{par}(S) = \mathrm{T}_{split}(S) + \mathrm{T}^{par}(\frac{S}{b}) + \mathrm{T}_{merge}(S) + 2\mathrm{T}_{com}(S)$$

Suppose that the size of the original problem is $S = b^q$ and problems of size $\widehat{S} = b^c$ are solved sequentially; then

$$\mathrm{T}^{par}(b^q) = \sum_{i=c+1}^{q} \left(\mathrm{T}_{split}(b^i) + \mathrm{T}_{merge}(b^i) + 2\mathrm{T}_{com}(b^{i-1}) \right) + \mathrm{T}^{seq}(b^c) \quad (2.2)$$

2.5.2 Granularity

Theoretically, the execution time of a "good" parallel program should decrease as the number of processors available for computation increases. The actual speedup of a program, however, is limited by the communication speed. Efficiency of a parallel algorithm is not determined solely by whether the algorithm uses all available resources or whether it uses them as soon as possible.

Parallel Base-Case Size

Communication overhead will make the parallel Divide-and-Conquer algorithm for problems of a certain size less efficient than the sequential algorithm. Let `ParBaseCaseSize` be the largest size of a problem such that the sequential algorithm is more efficient for its solution than the parallel one. Knowing the performance characteristics of the system, one can predict the value of `ParBaseCaseSize`.

Infinite number of processors. Suppose that the target architecture consists of an infinite number of identical processors ($N_p = \infty$). Suppose that each process is mapped onto a separate processor as described in section 2.5.1. Let T_k^{par} denote the execution time of the parallel implementation in which subproblems in the first k levels of the data flow graph are solved the using parallel algorithm, and subproblems on the lower levels are solved using the sequential algorithm. For example, T_1^{par} corresponds to the implementation in which the original problem is split into subproblems, which are then solved using the sequential algorithm on separate processors, and their subsolutions are then merged. T_k^{par} can be expressed as:

$$\mathrm{T}_k^{par}(S) = \sum_{i=1}^{k} \left(\mathrm{T}_{split}(\frac{S}{b^{i-1}}) + \mathrm{T}_{merge}(\frac{S}{b^{i-1}}) + 2\mathrm{T}_{com}(\frac{S}{b^i}) \right) + \mathrm{T}^{seq}(\frac{S}{b^k}) \quad (2.3)$$

Suppose that there exists a problem size $\widehat{S}$ such that the sequential solution of problems of size $\widehat{S}$ is more efficient than their parallel solution

with one level of the graph executed in parallel:

$$\mathrm{T}^{seq}(\widehat{S}) \leq \mathrm{T}_1^{par}(\widehat{S})$$

Using equations (2.3) and (2.1) we can find a condition on the size $\widehat{S}$ in terms of execution time of sequential algorithm and communication overhead:

$$\mathrm{T}^{seq}(\frac{\widehat{S}}{b}) \leq \frac{2}{K-1}\mathrm{T}_{com}(\frac{\widehat{S}}{b}) \tag{2.4}$$

For non-decreasing execution time functions T_{split} and T_{merge} and constant function $\mathrm{T}_{com}(S)$ it is possible to show that from inequality (2.4) follows that:

$$\forall n = 1, 2, \ldots : \ \mathrm{T}_n^{par}(\widehat{S}) \geq \mathrm{T}^{seq}(\widehat{S}),$$

Thus, if inequality (2.4) holds, then solving a problem of size $b\widehat{S}$ with any number of parallel steps is less efficient than its sequential solution. Even though in most distributed systems communication time, T_{com}, is not a constant, its dependence on the size of the message is usually relatively small (in comparison with functions T_{split} and T_{merge}). Therefore, for such systems the inequality (2.4) can be used as a good upper bound on the value of `ParBaseCaseSize`.

Finite number of processors. If the system consists of a *finite* number of processors of the same type, it imposes more restrictions on the value of `ParBaseCaseSize`. Let us find an upper bound on `ParBaseCaseSize` for the system in which the number of processors (N_p) is a power of K, i.e., $N_p = K^p$ for some p.

If the mapping strategy discussed in section 2.5.1 is used, then until the execution of p-th level of the tree, there will be only one process per processor. If the size of the original problem is b^q, then at level p each processor will be solving a subproblem of size $S' = b^{q-p}$. If the problems of size S' are then solved in parallel (with workload distributed equally between the processors), then each processor will have to execute the same total number of sequential procedures as if it were solving one problem of size S' sequentially. However, in addition to communication overhead, some system overhead will be added due to context switching. Thus, it is more efficient to solve problems of size S' sequentially than by using the parallel algorithm.

Therefore, for a distributed system with some finite number N_p of identical processors, the parallel base-case size for the original problem of size

S is determined by the performance parameters of the system and by the number of processors:

$$\begin{aligned} \mathrm{T}^{seq}(\widehat{S}) &\leq \frac{2}{K-1}\mathrm{T}_{com}(S) \qquad (2.5) \\ \texttt{ParBaseCaseSize} &= \max\{b\widehat{S}, \frac{S}{N_p}\} \end{aligned}$$

Optimal Depth of Data Flow Subtree

Because of the strategy behind the Divide-and-Conquer Archetype, the split of several levels of subproblems or the merge of several levels of subsolutions can be easily combined in one sequential process (see figure 2.3). Let us call such processes `Group_Split` and `Group_Merge`. By changing the depth of the data-flow subtree executed by one process, the user can control the granularity of a parallel Divide-and-Conquer algorithm.

If $t(s)$ is the execution time of one node in the graph (or ther `Split`() or `Merge`() procedure), then the execution time of the process that executes d levels of the tree is

$$T_{group}(t, d, S) = \sum_{i=0}^{d-1} K^i t(\frac{S}{b^i})$$

The optimal depth of the data-flow subtree achieving the best performance can be found by solving the minimization problem. Suppose that the optimal depth is equal to some constant D. Then, using the execution times of `Split`(), `Merge`() and `BaseCaseSolution`(), we derive an expression for the execution time of such an algorithm with depth D for original problem of size S, $\mathrm{T}(D, S)$, and find the value of D by solving the minimization problem:

$$\min_d \mathrm{T}(d, S)$$

where $\mathrm{T}(d, S)$ is:

$$\begin{aligned} \mathrm{T}(d, S) = & \sum_{i=1}^{d_1-1} \left(T_{group}(\mathrm{T}_{split}, d, \frac{S}{b^{id}}) + T_{group}(\mathrm{T}_{merge}, d, \frac{S}{b^{id}}) \right) \\ & + T_{group}(\mathrm{T}_{split}, d_2, \frac{S}{b^{id}}) + T_{group}(\mathrm{T}_{merge}, d_2, \frac{S}{b^{id}}) \\ & + \mathrm{T}^{seq}(\texttt{ParBaseCaseSize}) \end{aligned}$$

where $N_p = d_1 d + d_2$, and $d_2 < d$.

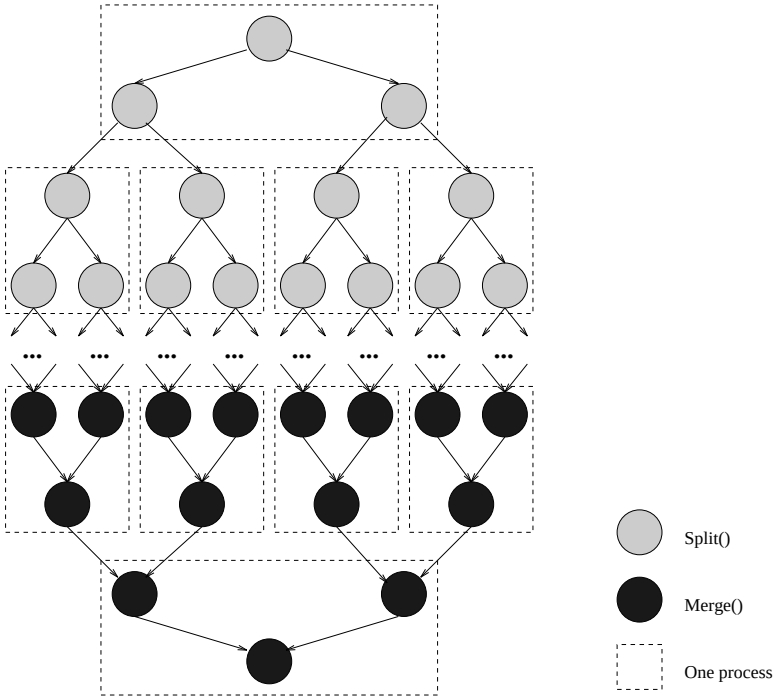

Figure 2.3: Group processes.

Figure 2.4 illustrates how the value of the depth of the subtree can affect the execution time of the parallel algorithm. The graph shows the execution time of two implementations of finding the minimum element in the array: the upper curve corresponds to an implementation with $D = 1$, and the lower curve corresponds to an implementation with D computed using the suggested approach.

2.5.3 Skeleton

The Data Flow approach with mapping described in section 2.5.1 has the structure shown below. Constant `Depth` denotes the depth of the data-flow subtree mapped onto one process. Procedures `Group_Split()` and `Group_Merge()` denote the group processes discussed in the previous section.

```
Solution_t Parallel_DnC1(Problem_t aProblem)
```

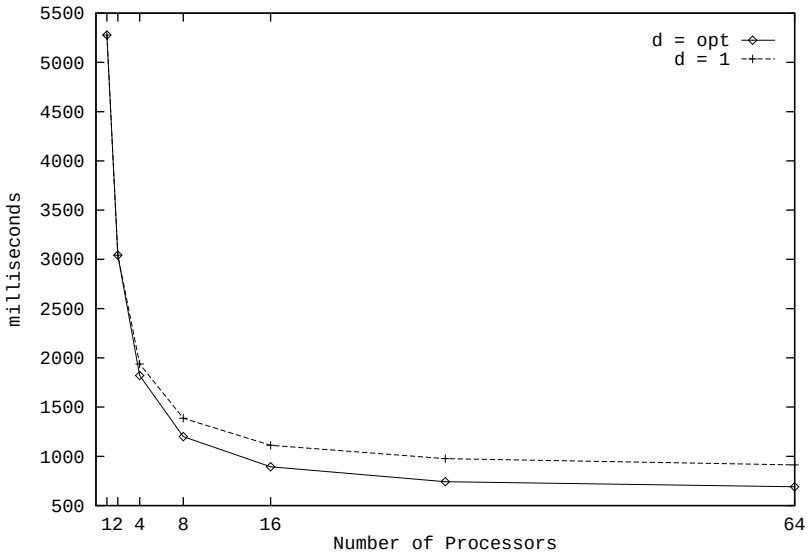

Figure 2.4: Execution time of finding the minimum element.

```
{
  Solution_t aSolution;

  if (Size(aProblem) ≤ ParBaseCaseSize)
      /* Problem is too small to be solved in parallel,
       * solve it sequentially */
    aSolsution = DnC(aProblem);
  else {
      /* solve problem in parallel */
    Problem_t subProblems[K^Depth];
    Solution_t subSolutions[K];
    OtherInfo_t Other[2^(Depth−1)];
    int i;

    Group_Split(Problem, subProblems, Other);
    parfor(i = 0; i < K^Depth; i++)
      subSolutions[i] = Parallel_DnC1(subProblems[i]);
```

```
    aSolution = Group_Merge(subSolutions, Other);
  }
}
```

2.6 Implementations

We used the approach presented in section 2.5 to implement software template in C with PVM, C with NX, and CC++. Since the code for all templates is very similar, we will present and discuss the source code for the CC++ implementation only.

A program is provided to the user to determine the optimal values of constants `Depth` and `BaseCaseSize`. The user is required to provide the parameters of the algorithm and the target system discussed in sections 2.3.1 and 2.5.

The user is required to define two classes `Problem_t` and `Solution_t` with the following public interface:

```
class Problem_t{
 public:
  Problem_t();
  ~Problem_t();

  int Size();
  void Split(Problem_t *);
  friend CCVoid& operator≪(CCVoid& , const Problem_t& );
  friend CCVoid& operator≫(CCVoid& , Problem_t& );
};

class Solution_t{
 public:
  Solution_t(){};
  ~Solution_t();

  void BaseCaseSolution(Problem_t& );
  void Merge(Solution_t *);
  friend CCVoid& operator≪(CCVoid& , const Solution_t& );
  friend CCVoid& operator≫(CCVoid& , Solution_t& );
};
```

The last two functions in each class are data transfer functions required by CC++ [12]. They define how the data of an object of the class should be transferred from one processor object to another. These functions are invoked when the objects of a class are used as arguments to procedures invoked on a remote processor object.

The CC++ implementation defines processor object type `DnC_t` which uses two classes `Machines_t` and `Tree_t`. These classes are provided to the user.

Class `Machines_t`: is used for representing and manipulating arrays of the names of the computers (processors) used in the computation. This class has the following public interface:

```
typedef char *string;

class Machines_t{
 public:
  int number;
  int *length;
  string *names;

  Machines_t();
  Machines_t(int n);
  Machines_t(int n, char **n1);
  ~Machines_t();

  void Part(int number_of_parts, int which_part, Machines_t& m);
  friend CCVoid& operator≪(CCVoid&, const Machines_t&);
  friend CCVoid& operator≫(CCVoid&, Machines_t&);
};
```

Member function `Part` is used to split the array of machine names into the given number of subarrays, and assign given part number to the array of machines `m`.

Class `Tree_t`: is used by procedures `Group_Split()` and `Group_Merge()` for storing and manipulating the tree of subproblems and subsolutions mapped onto one processor object.

```
typedef struct tree_level_t {
```

```
  int size;
    // the number of nodes on the level
  Problem_t *subproblems;
    // array of the subproblems of the level
  Solution_t *subsolutions;
    // array of the subsolutions of the level
    // Pairs subproblems[i] and subsolutions[i] form
    // the nodes of the level
  int *child_index;
    // array of indicies of the nodes of the lower level
    // connected to the nodes of the level
} tree_level_t;

class Tree_t{
 private:
  tree_level_t *Levels;
     // arrays of the pointers to the levels of the tree
  int NumberOfLevels;
     // number of levels in the tree
  int NumberOfChildren;
     // maximum number of children for a node

 public:
  Tree_t(int levels = 1, int numberofchildren = 2);
  ~Tree_t();
  tree_level_t * Level(int number);
  tree_level_t * LastLevel();
  void CreateNewLevel(int level);
};
```

Processor Object Type `DnC_t`

Class `DnC_t` has the following interface:

```
global class DnC_t{
 private:
  Machines_t Processors;
  int BaseCaseSize;
  int ParBaseCaseSize;
  int Depth;
```

```
  int Number_Of_Children;

  void Group_Split(Problem_t Problem, Tree_t *Tree);
  Solution_t Group_Merge(Tree_t *Tree);

 public:
  DnC_t(int basecasesize, int par_basecasesize, int splitdepth,
      int numberofchildren, Machines_t processors);
  Solution_t DnC(Problem_t problem);
  Solution_t Par_DnC(Problem_t problem);
};
```

Selected Member Functions

Function `Par_DnC()` follows the skeleton presented in section 2.5.3 very closely and does not requre any additional explanations.

```
Solution_t DnC_t::Par_DnC(Problem_t aProblem)
{
  if ((Processors.number == 1) ||
      (aProblem.Size() ≤ BaseCaseSize))
    return Divide_and_Conquer(aProblem);
  else {
    Solution_t aSolution;
    Tree_t * Tree = new Tree_t(SplitDepth+1,
                               Number_Of_Children);
    tree_level_t *last;
    DnC_t*global Children[Processors.number];
    int number_of_subproblems, part, k, i;

      // split the problem into at most Processors.number
      // of subproblems of size at most ParBaseCaseSize
    Group_Split(aProblem, Tree);

    last = Tree→LastLevel(SplitDepth);
    number_of_subproblems = last→size;
    assert(number_of_subproblems ≤ Processors.number);

      // each subproblem will be solve at >= 1  processor
```

```
    part = Processors.number / number_of_subproblems;
    parfor (int k = 0; k < number_of_subproblems; k++) {
      Machines * M = new Machines();
      Processors.Part(number_of_subproblems, k, *M);
      {
        proc_t placement = proc_t("DnC.out", M->names[0]);

        Children[k*part] = new (placement)
          DnC_t(BaseCaseSize, ParBaseCaseSize,
                SplitDepth, Number_Of_Children, *M);
        last->subsolutions[k] =
          Children[k*part].Par_DnC(last->subproblems[k]);

        delete Children[k*part];
      }
      delete M;
    }
    aSolution = Group_Merge(Tree);
    return aSolution;
  }
}
```

2.7 Applications

2.7.1 Mergesort

Problem Description: Given an array of N integers, sort the integers in ascending order.

Components:

Data types: Data types `Problem_t` and `Solution_t` for this problem are arrays of integers, and data type `OtherInfo_t` is undefined:

```
typedef struct {
   int size;            /* number of elements in the array */
   int *values;         /* the array of elements           */
} Problem_t, Solution_t;
```

Predicate IsSolution

$$isSolution(\texttt{P},\texttt{S}) \;=\; \texttt{P.size} = \texttt{S.size}$$
$$\text{and } \texttt{S.values} \text{ is some permutation of } \texttt{P.values}$$
$$\text{and } (\forall i : 0 < i < n : \texttt{S.values}[i-1] \leq \texttt{S.values}[i])$$

BaseCaseSize $\texttt{BaseCaseSize} = 1$

Functions and Procedures

```
int Size(Problem_t P)
{
  return P.size;
}


Solution_t BaseCaseSolution(Problem_t aProblem)
{
  return aProblem;
}

void Split(Problem_t aProblem,
           Problem_t subProblems[],
           OtherInfo_t Other)
{
  subProblems[0].size = aProblem.size/2;
  subProblems[1].size = aProblem.size - subProblems[0].size;
  subProblems[0].values = aProblem.values;
  subProblems[1].values = aProblem.values
    + subProblems[0].size;
}


Solution_t Merge(Solution_t subSolutions[],
                 OtherInfo_t Other)
{
  int i, j;
  Solution_t Solution;

  Solution.size = subSolutions[0].size +
    subSolutions[1].size;
  i = 0; j = 0;
  while ((i < subSolutions[0].size) &&
         (j < subSolutions[1].size))
```

```
    if ((subSolutions[0].array[i]) ≤
        (subSolutions[1].array[j])) {
      Solution.array[i+j] = subSolutions[0].array[i];
      i++;
    }
    else {
      Solution.array[i+j] = subSolutions[1].array[j];
      j++;
    }
  for (; i < subSolutions[0].size; i++)
    Solution.array[i+j] = subSolutions[0].array[i];
  for (; j < subSolutions[1].size; j++)
    Solution.array[i+j] = subSolutions[0].array[i];
  return Solution;
}
```

Parameters: function `Split()` divides the given array into two arrays of approximately equal size; thus, the algorithmic parameters of the merge-sort are $K = 2$ and $b = 2$. Using linear regression it is possible to find execution time functions for procedures `BaseCaseSolution()`, `Split()` and `Merge()`. On the Touchstone Delta these functions (in milliseconds) and the communication time function are:

$$\begin{aligned} \mathrm{T}_{base-case}(n) &= 0.0074 \\ \mathrm{T}_{split}(n) &= 0.0006 \\ \mathrm{T}_{merge}(n) &= 0.00075n + 0.00696 \\ \mathrm{T}_{com}(n) &= 0.00116n + 0.24757 \end{aligned}$$

where n is the size of the array.

Performance results: Using the above parameters we can compute the optimal depth and the value of `ParBaseCaseSize`. The value of `ParBaseCaseSize` is 128.

Table 2.1 summarizes execution time of the algorithm on the Touchstone Delta for different values of the depth. Execution time is given in seconds. The numbers that correspond to the depth chosen by the performance model as optimal are shown in *emphasis* font. The discrepancy between predicted optimal depth and actual optimal depth can be explained by the fact that the global clock was used in all performance measurements (including the ones used for computing T_{split}, T_{com}, etc) and by the fact that the performance model does not take into account such parameters as contention in the network, memory caching, and the overhead of recursive calls.

Number of processors	d = 1	d = 2	d = 3	d = 4	d = 5
2	*10.4*	—	—	—	—
4	5.96	*6.02*	—	—	—
8	3.79	*3.90*	4.13	—	—
16	2.71	*2.82*	3.06	3.41	—
32	2.20	*2.31*	2.55	2.89	3.28
64	1.94	*2.05*	2.35	2.65	3.03

Table 2.1: Execution time of the mergesort algorithm on Touchstone Delta (in seconds).

The graph of speedup of the parallel mergesort with respect to the sequential algorithm is shown in figure 2.5. The parallel mergesort was implemented using the depth predicted by the model.

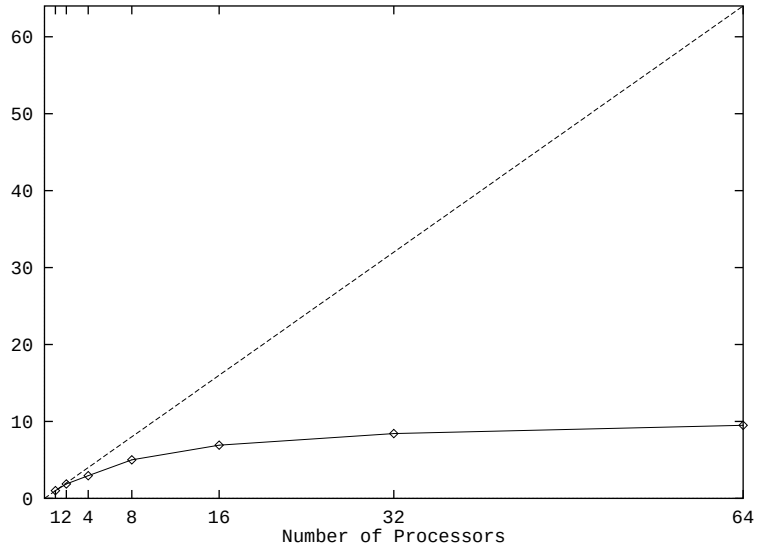

Figure 2.5: Speedup of parallel mergesort on Touchstone Delta.

Chapter 3

The Branch and Bound Archetype

3.1 Introduction

Branch and Bound is a technique for searching an implicit directed graph which is usually acyclic or even a tree [2].

The Branch and Bound approach is often used for finding an optimal solution to some problem specified by a finite but possibly very large space of solutions. The search graph for such a problem consists of nodes corresponding to a partition of the solution space, with successive nodes representing smaller and smaller subpartitions of preceding nodes. For each node a bound on the possible value of any solution within the partition of this node is calculated. Usually, this bound is used to prune certain branches of a search tree if a better solution has been already found. Sometimes, a depth-first search or a breadth-first search strategy is used. More often, however, the calculated bound is also used to choose which of the open nodes of the tree should be explored first.

Because of the unstructured search strategy, a simple parallel implementation of the Branch and Bound algorithms often does not scale very well, because processes exploiting some branches of the tree do not have current information about the best solution found so far. One could improve the performance of the program if a parallel Branch and Bound implementation is composed with some heuristic algorithm for solving the problem. The suboptimal solutions found by the heuristic algorithm can then be used together with the calculated bounds for pruning certain branches of the search

tree.

3.1.1 Assumptions

For the purposes of this report we make following assumptions:

- the problem being solved is a maximization problem that can be specified by problem type, objective function being maximized and a number of constraints,
- the bound of a partition is a single real number.

If these assumptions do not hold, the overall design approach described in this chapter is still valid; only small number of details has to be changed. Moreover, since any minimization problem can be easily converted into a maximization problem (by changing the sign of the objective function), the above assumptions hold for most problems that can be solved using the Branch and Bound approach.

3.2 Archetype Skeleton

It has been suggested in many books on computer algorithms (e.g., [2]) that the set of open partitions — partitions yet to be expanded — should be stored in a heap. In the program that follows we use data type `Heap_t`, with the following interface:

```
void AddPartition(Heap_t , Partition_t );
int Empty(Heap_t );
Partition_t RemoveBestPartition(Heap_t );
void RemoveWorseThan(Heap_t , Partition_t );
```

Using this data type the skeleton of the Branch and Bound Archetype is as follows:

```
Partition_t BnB(Partition_t OriginalPartition)
{
  Heap_t UnexpandedPartitions;
  Partition_t SubPartitions[K];
  Partition_t BestSolution;
  Partition_t aPartition;
  int n, i;
```

```
  AddPartition(UnexpandedPartitions, OriginalPartition);
  while (!Empty(UnexpandedPartitions)) {
    aPartition = RremoveBestPartition(UnexpandedPartitions);
    n = Branch(aPartition, SubPartitions);
    for (i = 0; i < n; i++) {
      if (isSolution(SubPartitions[i]) &&
          (Bound(SubPartitions[i]) > Bound(BestSolution))) {
        BestSolution = SubPartitions[i];
        RemoveWorseThan(UnexpandedPartitions, BestSolution);
      }
      else
        if (Bound(SubPartitions[i]) > Bound(BestSolution))
          AddPartition(UnexpandedPartitions,
                       SubPartitions[i]);
    }
  }
  return BestSolution;
}
```

Constant `K` is the maximum number of subpartitions returned by the function `Branch()`.

3.3 Archetype Components

In order to develop a Branch and Bound algorithm, the user has to define one data type and several functions on that data type, as described below:

Data type: `Partition_t` is a user-defined data type representing a partition in the space of feasible solutions, i.e., a non-empty set of feasible solutions.

Predicates and Ghost Functions: For the purposes of specification and reasoning, some predicates and ghost functions should be defined as follows:

$Value(S)$ is a number-valued function that returns the value of the objective function for given *feasible* solutions.

$Set(P)$ is a function whose value is the set of all feasible solutions for partition P.

$Metric(P)$ is a number-valued function, whose value corresponds to some metric of the partition, such that $Metric(P) = 0$ implies that partition P contains only one feasible solution, that is $|Set(P)| = 1$. Functions $Set(P)$ and $Metric(P)$ are related to each other as follows:

$$\begin{array}{rcl} (Set(P_1) \subseteq Set(P_2)) & \equiv & (Metric(P_1) \leq Metric(P_2)) \\ & \text{and} & \\ (Set(P_1) \subset Set(P_2)) & \Rightarrow & (Metric(P_1) < Metric(P_2)) \end{array}$$

$isWellFormed(P)$ is a predicate that holds if and only if given partition P is a well-formed partition. This predicate is required because in programming languages variables of the certain types are allowed to take many values, some of which do not correspond to a valid partition within the context of the problem being solved.

Procedures and Functions:

`Branch()` function divides a given partition into 1 or more subpartitions with strictly smaller metric value. Formally, the `Branch()` function is defined as follows:

```
int Branch(Partition_t aPartition,
           Partition_t subPartitions[])
/* Precondition:  isWellFormed(aPartition)
 * Postcondition: (Return value = n) and (n ≥ 1) and
 *  (∀ i: 0 ≤ i < n:
 *            Metric(subPartitions[i]) < Metric(aPartition))
 *  and (Set(aPartition) = ⋃_{i=0}^{n-1} Set(subPartitions[i]))
 */
```

Note that this specification can be weakened if we use the set of all feasible solutions to the *original* problem being solved, $\mathcal{P}$. Then the specification of function `Branch()` is as follows:

```
int Branch(Partition_t aPartition,
           Partition_t subPartitions[])
/* Precondition:  isWellFormed(aPartition)
 * Postcondition: (Return value = n) and (n ≥ 1) and
```

$*$ $(\forall\ i:\ 0 \leq\ i < n:$
$*$ $\quad Metric(subPartitions[i]) < Metric(aPartition))$
$*$ $and\ (Set(aPartition) \subseteq \bigcup_{i=0}^{n-1} Set(subPartitions[i]))$
$*$ $and\ (\forall\ i: 0 \leq\ i < n: Set(subPartitions[i]) \subset Set(\mathcal{P}))$
$*/$

`isSolution()` is a function that tests whether a given partition contains one or more feasible possible solution.

```
int isSolution(Partition_t aPartition)
/* Precondition:  isWellFormed(aPartition)
 * Postcondition:
 *   ((Return value = 0) and (|Set(aPartition)| > 1))
 *   or ((Return value = 1) and (|Set(aPartition)| = 1))
 */
```

`Bound()` is a user-defined function that evaluates the bound on all feasible solutions within a given partition. If a partition contains only one solution (or equivalently, function `isSolution()` returns 1 for a given partition), then function `Bound()` returns the true solution value for the only solution in the partition; otherwise, `Bound()` returns some upper bound on all possible solutions within a given partition.

```
float Bound(Partition_t aPartition)
/* Precondition:  isWellFormed(aPartition) and
 *       (|Set(aPartition)| > 1)
 * Postcondition: (Return value = f) and
 *       (∀ s: s ∈ Set(aPartition): Value(s) ≤ f)) and
 *       ((|Set(aPartition)| = 1) ⇒
 *                   (∀s: s ∈ Set(aPartition): Value(s) = f))
 */
```

The efficiency of a Branch and Bound algorithm depends on the tightness of the bound returned by the `Bound()` function.

3.4 Approaches to Parallel Implementation

Any parallel approach to the best-first search strategy can be used to implement a parallel Branch and Bound algorithm. Many approaches that

have been already described and analyzed (for example, see [6, 9]) can be implemented as programming templates using the components described in section 3.3.

One of the simplest centralized strategies is the master-and-slave strategy. One processor is assigned the role of "master" and stores the list of unexpanded partitions. Other processors are "slaves" that expand the partitions sent by the "master" process, calculate their bounds and return the results to the "master". At each instant, when a "slave" becomes idle the "master" selects the best partition from the list of unexpanded partitions and sends it to the "slave" process. Since in this strategy several partitions are expanded at once, the parallel implementation may expand nodes that would not be expanded by a sequential algorithm.

The performance of the master-and-slave strategy is limited by the fact that a message is exchanged between the "master" and a "slave" processes for each partition. This factor can affect the scalability of the parallel implementation. Several small modifications to the strategy can be made to improve performance:

- Together with the partition to be expanded, the "master" process sends the best currently known solution, thus delegating part of the pruning to the "slave" and reducing the number of messages in the system.
- Another way to reduce the number of messages in the system is to allow "slave" processes to expand partitions down to a certain level of the search graph.
- The master-and-slave implementation of a problem can be composed with some heuristic algorithm for solving the problem. Then, solutions found by the heuristic process can then be used to eliminate some paths of the search graph and avoid their expansion.

3.5 Implementations

The master-and-slave strategy and its several modifications were implemented in C with NX and C with PVM. Since the source code for the NX and PVM implementations is very similar, we will present and discuss the PVM implementation only.

The user is required to define data type `Partition_t`, several functions on that data type, and global data. In addition to the archetype's compo-

nent functions, the user has to define procedure `FreePartition()` that deallocates the memory that was allocated for a partition, and several platform-dependent communication procedures.

```
   /* Archetype's Components */
int Branch(Partition_t aPartition,
           Partition_t *subPartitions);
float Bound(Partition_t aPartition);
int IsSolution(Partition_t aPartition);

  /* Memory Management Procedure */
void FreePartition(Partition_t aPartition);

   /* Communication Procedures,  */
void SendGlobalData(long msg_type, long node);
void ReceiveGlobalData(long msg_type);
void SendPartition(Partition_t aPartition,
                   long msg_type, long node);
void ReceivePartition(Partition_t * aPartition,
                      long msg_type);
```

Slave

Until a termination message is received from the "master" process, a "slave" receives a partition to expand, divides it into several subpartitions using function `Branch()`, calculates the bounds for each of the subpartitions, and sends the results back to the master process, together with a request for another partition.

```
void Slave(int MaxNumOfSubPartitions)
{
  Partition_t p, *subpartitions;
  float bound, solution;
  int terminate = 0;
  int i, n;
  int master, dummy;
  int bufid, bytes, msg_type, tid, mytid;

    /*** Initialize local variables ***/
  mytid = pvm_mytid();
  master = pvm_parent();
  subpartitions = (Partition_t *)calloc(sizeof(Partition_t),
```

```
                                        MaxNumOfSubPartitions);

    /*** receive global data ***/
  pvm_recv(master, MSG_GLOBAL_DATA);
  ReceiveGlobalData(MSG_GLOBAL_DATA);

  while (!terminate) {
      /*** wait for the next message to arrive ***/
    bufid = pvm_recv(master, -1);
    pvm_bufinfo(bufid, &bytes, &msg_type, &tid);

    switch(msg_type) {
      case MSG_TERMINATE:                          /* termination detection */
        pvm_upkint(&dummy, 1, 1);
        terminate = 1;
        break;
      case MSG_PARTITION:                           /* a partition received */
        pvm_upkfloat(&solution, 1, 1);
        ReceivePartition(&p, msg_type);

        n = Branch(p, subpartitions);
        for (i = 0; i < n; i++) {
          bound = Bound(&(subpartitions[i]));
          if (IsSolution(subpartitions[i]))
            msg_type = MSG_A_SOLUTION;
          else
            msg_type = MSG_PARTITION;
          pvm_initsend(PvmDataDefault);
          pvm_pkfloat(&bound, 1, 1);
          SendPartition(subpartitions[i],
                        mytid*10+msg_type, master);
          pvm_send(master, msg_type);
          FreePartition(subpartitions[i]);
        }
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&dummy, 1, 1);
        pvm_send(master, MSG_REQUEST);
        FreePartition(p);
        break;
      }
  }
}
```

Master

In the following implementation, the "master" process takes as an argument an array of task id's of "slave" processes. It assumes that initially all "slave"

processes are idle and are waiting for a partition to be sent. After global information is transferred to the slave processes, the "master" sends off the first partition. Depending on the type of the message received, the "master" takes various actions: registers a received solution, inserts a received partition into the list of unexpanded partitions, sends another partition to the "slave" process or registers "slave" process as idle. Whenever the received solution is better than the current best solution, the "master" removes partitions with lower bounds from the list of unexpanded partitions. When all "slave" processes are registered as idle and the list of unexpanded partitions is empty, the "master" process terminates and sends termination messages to all "slave" processes.

```
Partition_t Master(Partition_t aPartition,
                   int numWorkers, int *Workers)
{
  Node_t *Root = NULL;
  Partition_t partition;
  Partition_t aSolution;
  float bound, solution = -1.0;
  int found_a_solution = 0;
  int first, last, numIdleWorkers, dummy;
  int terminate = 0, i;
  int msg_type, bufid, bytes, tid;

    /*** Initialize local variables ***/
  first = 0;
  last = numWorkers-1;
  numIdleWorkers = numWorkers;

    /*** Send global data off to all workers ***/
  for (i = 0; i < numWorkers; i++) {
    pvm_initsend(PvmDataDefault);
    SendGlobalData(MSG_GLOBAL_DATA, Workers[i]);
    pvm_send(Workers[i], MSG_GLOBAL_DATA);
  }

    /*** Send the first partition off to the first worker ***/
  pvm_initsend(PvmDataDefault);
  pvm_pkfloat(&solution, 1, 1);
  SendPartition(aPartition, MSG_PARTITION, Workers[first]);
  pvm_send(Workers[first], MSG_PARTITION);
  first = (first+1) % numWorkers;
  numIdleWorkers--;

    /*** continue until there are no more partitions to expand
     *** and all workers are idle ***/
```

```
while ((!isEmpty(&Root)) || (numIdleWorkers < numWorkers)) {
    /*** wait for a message to arrive ***/
  bufid = pvm_recv(-1, -1);
  pvm_bufinfo(bufid, &bytes, &msg_type, &tid);

  switch (msg_type) {
    case MSG_A_SOLUTION:                    /* message contains a solution */
      pvm_upkfloat(&bound, 1, 1);
      ReceivePartition(&partition, tid*10+msg_type);
      if ((!found_a_solution) ||
          ((found_a_solution) && (bound > solution))) {
        if (found_a_solution)
          FreePartition(aSolution);
        aSolution = partition;
        solution = bound;
        RemoveBadNodes(&Root, solution);
        found_a_solution = 1;
      }
      else
        FreePartition(partition);
      break;
    case MSG_REQUEST:               /* message contains a request for work */
      pvm_upkint(&dummy, 1, 1);
      if (!isEmpty(&Root)) {
        partition = RemoveFirst(&Root);
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&solution, 1, 1);
        SendPartition(partition, MSG_PARTITION, tid);
        pvm_send(tid, MSG_PARTITION);
        FreePartition(partition);
      }
      else {
        last = (last + 1) % numWorkers;
        Workers[last] = tid;
        numIdleWorkers++;
      }
      break;
    case MSG_PARTITION:                     /* message contains a partition */
      pvm_upkfloat(&bound, 1, 1);
      ReceivePartition(&partition, tid*10+msg_type);
      if ((!found_a_solution) ||
          ((found_a_solution) && (bound > solution)))
        if (numIdleWorkers > 0) {
          pvm_initsend(PvmDataDefault);
          pvm_pkfloat(&solution, 1, 1);
          SendPartition(partition, MSG_PARTITION,
                        Workers[first]);
```

```
            pvm_send(Workers[first], MSG_PARTITION);
            first = (first+1) % numWorkers;
            numIdleWorkers--;
            FreePartition(partition);
          }
          else
            InsertNode(&Root, partition, bound);
        else
          FreePartition(partition);
        break;
    }
  }

    /*** Send "terminate" message to all workers ***/
  for (i = 0; i < numWorkers; i++) {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&terminate, 1, 1);
    pvm_send(Workers[i], MSG_TERMINATE);
  }
  DeleteList(&Root);
  return aSolution;
}
```

3.6 Applications

3.6.1 Zero-One Knapsack

Problem Description: Given a knapsack of capacity C, and n objects with non-zero weights w_i and non-zero values v_i, find a collection of objects to be put into the knapsack that maximizes its total value, or

$$\begin{array}{rl} \text{maximize} & \sum_{i=0}^{n-1} v_i x_i \\ \text{subject to} & \sum_{i=0}^{n-1} w_i x_i \leq C \\ \text{where} & x_i \in \{0,1\} \end{array}$$

where $x_i = 1$ means that object i is included in the knapsack, and $x_i = 0$ means that it is not included in the knapsack.

Components: The problem being solved is defined by the values of the following global variables:

```
float Capacity;                 /* capacity of the knapsack */
int numObjects;                 /* number of objects        */
float ObjectWeights[numObjects]; /* objects' weights         */
```

```
float ObjectValues[numObjects];      /* objects' values          */
/* the objects are ordered in descending order according to the
 * value-density */
```

Data type: A partition is defined by the number of objects about which the decision has been made, that is, by the number of variables x_i whose values are fixed at 1 or 0, and by their values. Therefore, we define the data type `Partition_t` as follows:

```
typedef struct {
  int k;           /* number of fixed variables                 */
  char x[k];        /* values of the variables, 1 or 0          */
  float value;     /* sum: i in 0..(k-1): x[k]*ObjectValues[k]  */
  float weight;    /* sum: i in 0..(k-1): x[k]*ObjectWeights[k] */
} Partition_t;
```

Predicate and Ghost Functions: The objective function $Value(S)$ is defined only for partitions S such that S.`k = numObjects`:

$$Value(\texttt{S}) = \sum_{i=0}^{\texttt{numObjects}} \texttt{S.x}[i] \cdot \texttt{ObjectValues}[i]$$

The set of feasible solutions in a given partition `P`, $Set(\texttt{P})$, is formed by all possible combinations of `numObjects` values y_i such that:

$$\begin{aligned} & (\forall i : 0 \le i < \texttt{numObjects} : y_i \in \{0,1\}) \\ \text{and} \quad & (\forall i : 0 \le i < \texttt{P.k} : y_i = \texttt{P.x}[\texttt{i}]) \\ \text{and} \quad & \sum_{i=0}^{\texttt{numObjects}} y_i \cdot \texttt{ObjectWeight}[\texttt{i}] \le \texttt{Capacity} \end{aligned}$$

Finally, function $Metric(\texttt{P})$ is defined as follows:

$$Metric(\texttt{P}) = \texttt{numObjects} - \texttt{P.k}$$

Procedures: In order to calculate an upper bound on the value of the knapsack in partition `P`, the cheesecake problem is solved for objects $\texttt{P.k} + 1, \ldots, \texttt{numObjects}$ with capacity `Capacity` − `P.weight`:

```
float Bound(Partition_t aPartition) {
  float value, capacity;
  int i;

  value = 0.0;
  capacity = Capacity - aPartition.weight;
  for (i = aPartition.K;
       (capacity > 0.0) && (i < numObjects); i++)
    if (ObjectWeights[i] ≤ capacity) {
      value += ObjectValues[i];
      capacity -= ObjectWeights[i];
    }
    else {
      value += ObjectValues[i]*capacity/ObjectWeights[i];
      capacity = 0.0;
    }
  return (value+aPartition.value);
}

int isSolution(Partition_t aPartition) {
  return (aPartition.k == numObjects);
}
```

The function `Branch()` divides the partition `P` into at most two subpartitions by making decision about the object `P.k`. In one subpartition the object `P.k` is put into the knapsack, in the other one it is left out:

```
int Branch(Partition_t aPartition, Partition_t subPartitions[2]){
     /* copy the values of the fixed variables from aPartition */
  subPartitions[0] = aPartition;
  subPartitions[1] = aPartition;

     /* fix value of (aPartition.k)-th variable */
  subPartitions[0].k = aPartition.k + 1;
  subPartitions[1].k = aPartition.k + 1;

  /* do not put the object in the knapsack */
  subPartitions[0].x[aPartition.k] = 0;
  /* put the object into the knapsack (if possible) */
  if (ObjectWeights[aPartition.k] ≤
      subPartitions[1].capacity) {
    subPartitions[1].x[aPartition.k] = 1;
    subPartitions[1].value += ObjectValues[aPartition.k];
    subPartitions[1].weight += ObjectWeights[aPartition.k];
    return 2;
```

```
    }
    else
      return 1;
}
```

Performance Results: Figure 3.1 compares the execution time of the sequential zero-one knapsack problem with the execution time of the master-and-slave strategy (`ms` curve) and the master-and-slave strategy in which the "master" process sends the best known solution together with the partition to expand (`ms1` curve). The measurements were taken on the Touchstone Delta for 10,000 objects with uniformly distributed non-zero weights and values.

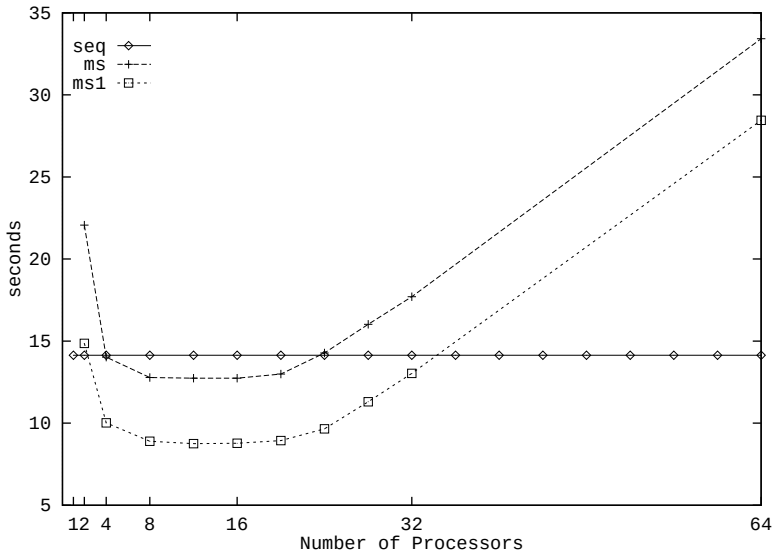

Figure 3.1: Execution time of zero-one knapsack program on Touchstone Delta.

The initial increase in execution time can be explained by the fact that when only 2 processors are used there is only one slave process. Therefore, the parallel implementation is computationally equivalent to the sequential one, except for communication overhead. The increase in execution time

for more than 20 processors can be explained by the fact that the "slave" processes spend some time expanding "bad" partitions and also by the increasing amount of communication, so that the master process becomes a bottleneck.

Chapter 4

Conclusion

We have presented two programming archetypes in combinatorics and optimization. For each archetype the program skeleton, archetype components and several implementations were given. For each archetype an example application from the archetype's domain was presented to illustrate how the archetype can be used.

It is interesting to note that once the template for an archetype had been written, the effort required for developing a parallel application was decreased significantly. In addition to the examples presented in this report several other example applications were developed: Manhattan Skyline, Nearest Neighbor, Traveling Salesman Problem and Zero-One Knapsack with two knapsacks.

Several directions for further development of the discussed Archetypes present themselves:

- A software template can be written for the Control Flow approach to parallel implementation of Divide-and-Conquer algorithms. The user might be required to develop sequential program in a specific fashion, so as to simplify the parallelization step.

- By using the Divide-and-Conquer archetype presented in chapter 2 the user can reduce the effort required for developing a parallel Divide-and-Conquer algorithm by developing and debugging the sequential program first. However, the scalability of the presented approach is far from perfect. A modified Divide-and-Conquer archetype with a more scalable parallel implementation would be even more useful. Such implementation is presented and discussed in [4].

- A performance model for the Branch and Bound archetype that can predict the performance of a parallel implementation or choose an efficient implementation for target architecture can be a wonderful tool for programmers.

Appendix A

Electronic Textbook

The full text of the programming templates in CC++, C and NX, and C and PVM, together with documentation, and several example programs in addition to the ones presented in this report will be made publicly available as part of the electronic textbook on Parallel Programming Archetypes. Several chapters of the textbook are currently available on the World Wide Web at `http://www.etext.caltech.edu`. The structure and contents of the textbook are described in [1].

Bibliography

[1] Paul Ainsworth and Svetlana Kryukova. A multimedia interactive environment using program archetypes: Divide-and-conquer. Technical Report Caltech-CS-TR-93-36, Computer Science Department, California Institute of Technology, 1993.

[2] Giles Brassard and Paul Bratley. *Algorithmics: theory and practice.* Prentice-Hall, Inc., 1988.

[3] P. Carlin, M. Chandy, and C. Kesselman. The compositional C++ language definition. Technical Report CS-TR-92-02, Computer Science Department, California Institute of Technology, 1992.

[4] K. Mani Chandy and Svetlana Kryukova. Parallel software architectures. Technical Report forthcoming, Computer Science Department, California Institute of Technology, 1995.

[5] M. Chandy, R. Manohar, B. Massingill, and D. Meiron. Integrating task and data parallelism with the collective communication archetype. Technical Report CS-TR-94-08, Computer Science Department, California Institute of Technology, 1994.

[6] J. Eckstein. Control strategies for parallel mixed integer branch and bound. In *Supercomputing '94 Proceedings*, pages 41–48. The Institute of Electrical and Electronics Engineers, Inc., 1994.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[8] Intel Corporation. *Touchstone Delta C System Calls Reference Manual*, 1991.

[9] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: design and analysis of parallel algorithms*, chapter Search Algorithms for Discrete Optimization Problems. The Benjamin/ Cummings Publishing Company, 1994.

[10] Berna L. Massingill. *Parallel Programming Archetypes in Scientific Computing* (working title). PhD thesis, Computer Science Department, Caltech, 1995. working title.

[11] Bernard M.E. Moret and Henry D. Shapiro. *Algorithms from P to NP - Volume I: Design and Efficiency*. The Benjamin/ Cummings Publishing Company, 1991.

[12] P. Sivilotti and P. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Computer Science Department, California Institute of Technology, 1994.